Unit-4

Syntax **D**irected **T**ranslation

BY:
TRUSHA R. PATEL
ASST. PROF.
CE DEPT., CSPIT, CHARUSAT

Introduction

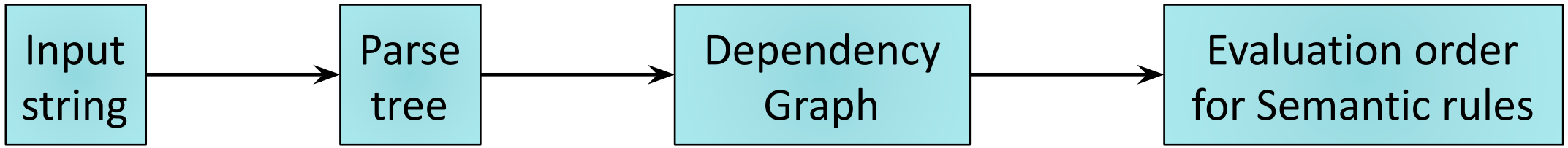
- 2 notations for associating semantic rules with production

  Syntax directed definition

  Translation scheme

- Using both (syntax directed definition & translation scheme) we parse the input steam, build the parse tree
  and then traverse the tree as needed to evaluate the semantic rules at the parse nodes
- Evaluation of the semantic rules may
  - Generate code
  - Save information in a symbol table
  - Issue error message
  - Perform any other activity

Conceptual view of Syntax Directed Translation



Syntax Directed Definition (SDD)

- SDD is generalization of a CFG in which each grammar symbol has an associated set of attributes
- Node of parse tree is grammar symbol
  then attributes holds its information
- An attribute can be anything
  A string / A number / A type / A memory location etc.
- The value of attribute is defined by semantic rule associate with the production used at that node

Syntax Directed Definition (SDD)

- Attribute can be of two types

(1) Synthesized attributes

- Value of synthesized attributes at a node is computed from the values of attributes at the children of that node

(2) Inherited attributes

- Value of inherited attributes is computed from the value of attributes at siblings and parent of that node

Syntax Directed Definition (SDD)

- Semantic rules set up dependencies between attributes that can be represented by a graph
- From dependency graph, we derive an evaluation order for the semantic rules
- Evaluation of semantic rules defines the values of the attributes at the nodes in the parse tree for the input string
- The parse tree showing values of attributes at each node is called “Annotated parse tree”
- The process of computing the attribute values at the node is called “Annotating parse tree” or “Decoding parse tree”

Form of SDD

- In SDD each grammar production $A \rightarrow \alpha$ has associated with it a set of semantic rules of the form $b = f(c_1, c_2, \ldots, c_k)$ where f is the function and either

  1. b is a synthesized attribute of A and
     $c_1, c_2, .. c_k$ are attribute belong to the grammar symbols of the production
     OR

  2. b is an inherited attribute of one of the grammar symbols on the right hand side of production and
     $c_1, c_2, .. c_k$ are attribute belong to the grammar symbols of the production

- In either case attributes b depends on attributes $c_1, c_2, .. c_k$

SDD = grammar + semantic rule

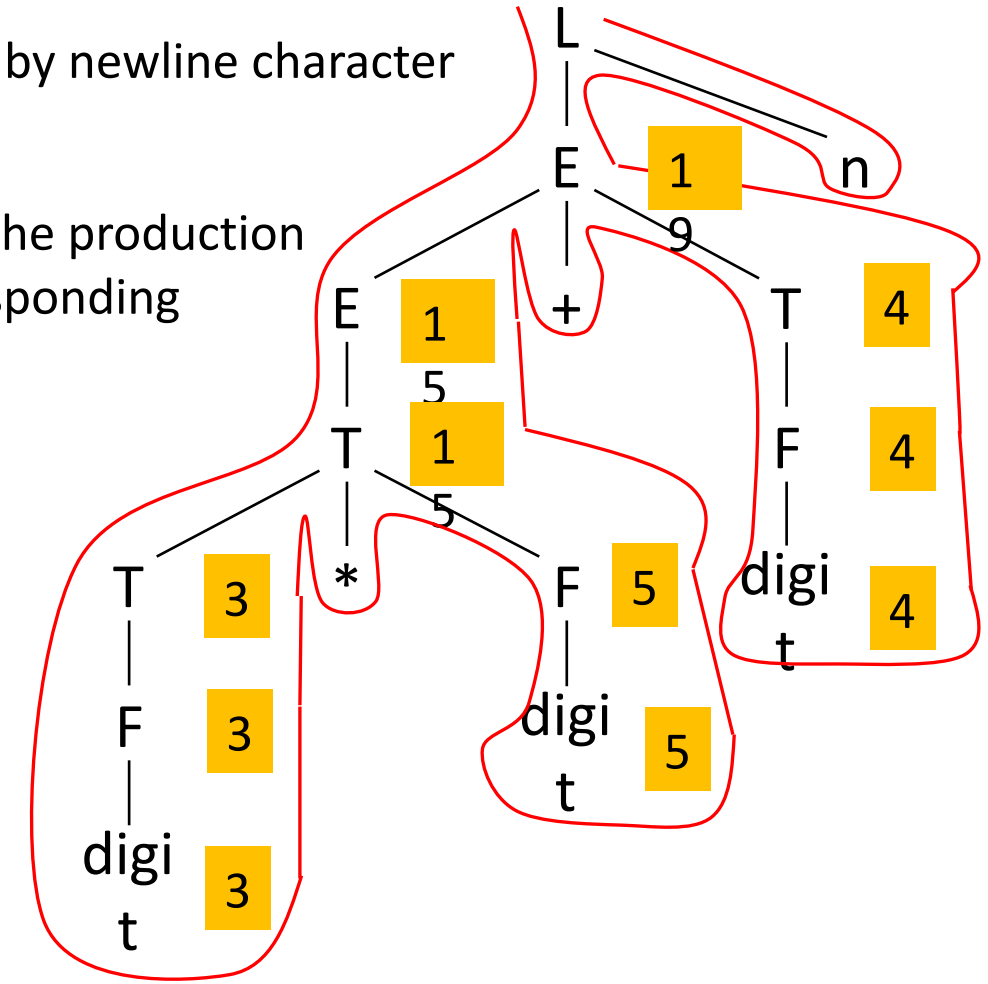
SDD (Simple Desk Calculator)

Production	Semantic Rule
L → E n	Print (E.val)
E → E + T	E.Val = E.val + T.val
E → T	E.val = T.val
T → T * F	T.val = T.val * F.val
T → F	T.val = F.val
F → (E)	F.val = E.val
F → digit	F.val = digit.lexval

Process string
3 * 5 + 4 followed by newline character

- Traverse tree
- When reduce the production perform corresponding semantic rule

Output :
19



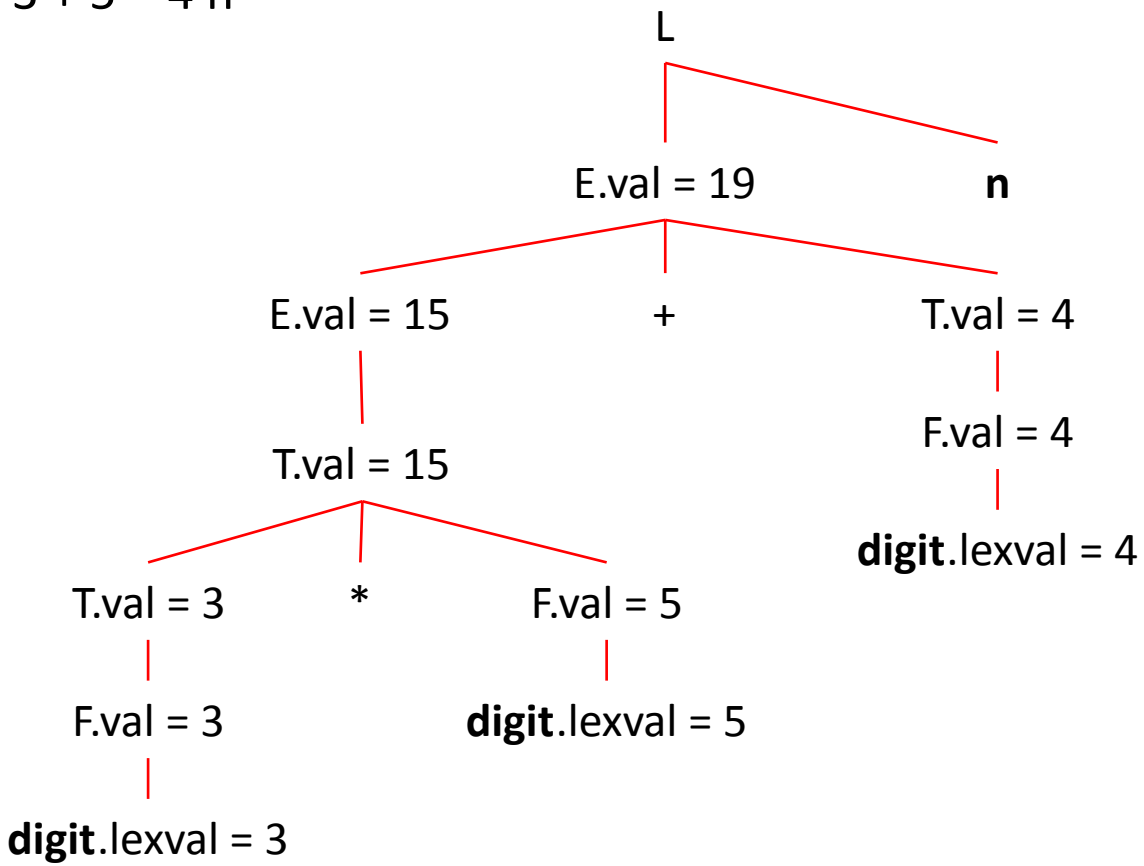
- It reads the input line containing an arithmetic expression involving digits, parenthesis, +, * and followed by new line character **n** and print the value of expression

SDD (Simple Desk Calculator)

Production	Semantic Rule
L → E n	Print (E.val)
E → E + T	E.Val = E.val + T.val
E → T	E.val = T.val
T → T * F	T.val = T.val * F.val
T → F	T.val = F.val
F → (E)	F.val = E.val
F → digit	F.val = digit.lexval

Example of Synthesized attribute

Annotated parse tree for
3 + 5 * 4 n



Output :
19

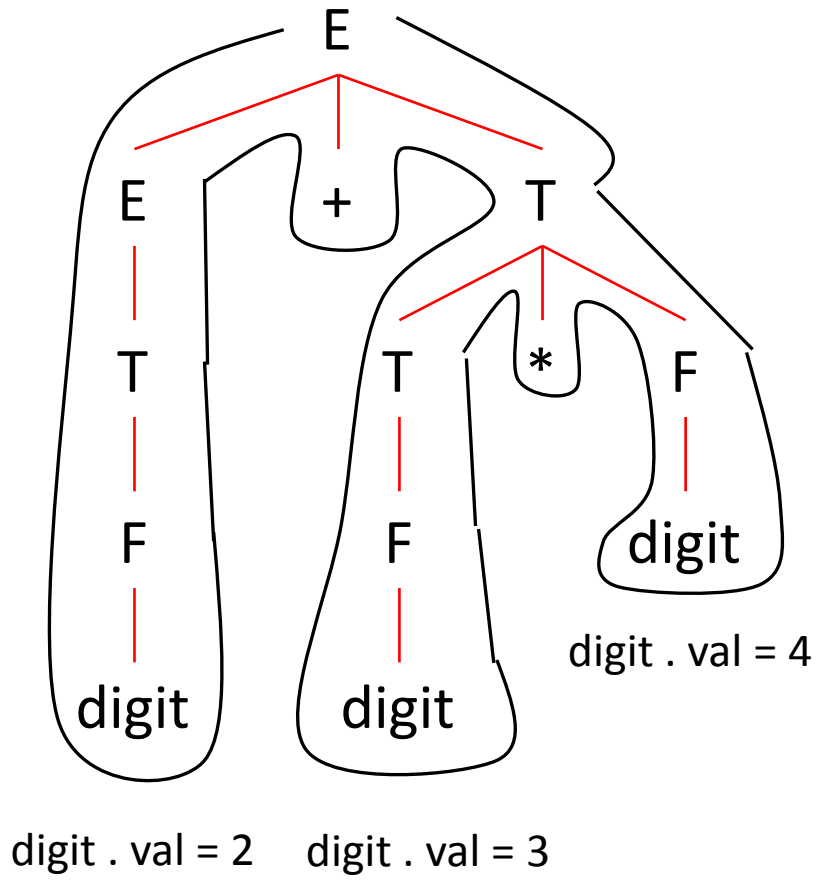
SDD (infix to postfix conversion)

Production	Semantic Rule
E → E + T	Print (+)
E → T	{}
T → T * F	Print (*)
T → F	{}
F → digit	Print (digit.val)

Process string
2 + 3 * 4

Output :

2 3 4 * +



SDD

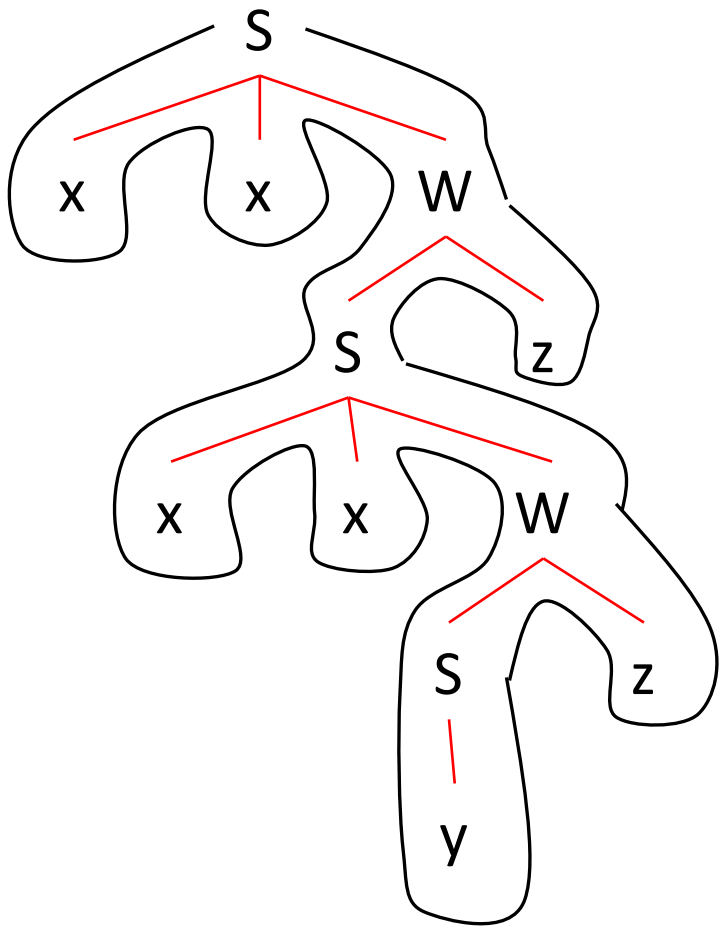
Production	Semantic Rule
S → xxW	Print (1)
S → y	Print (2)
W → Sz	Print (3)

Process string

x x x x y z z

Output :

2 3 1 3 1



SDD

Production	Semantic Rule	Semantic Rule	Semantic Rule
N → L	N . count = L . count	N . count = L . count	N . count = L . count
L → L B	L . count = L . count + B. count	L . count = L . count + B. count	L . count = L . count + B. count
L → B	L . count = B. count	L . count = B. count	L . count = B. count
B → 0	B . count = 0	B . count = 1	B . count = 1
B → 1	B . count = 1	B . count = 0	B . count = 1

Make SDD to count number of 1's in a string

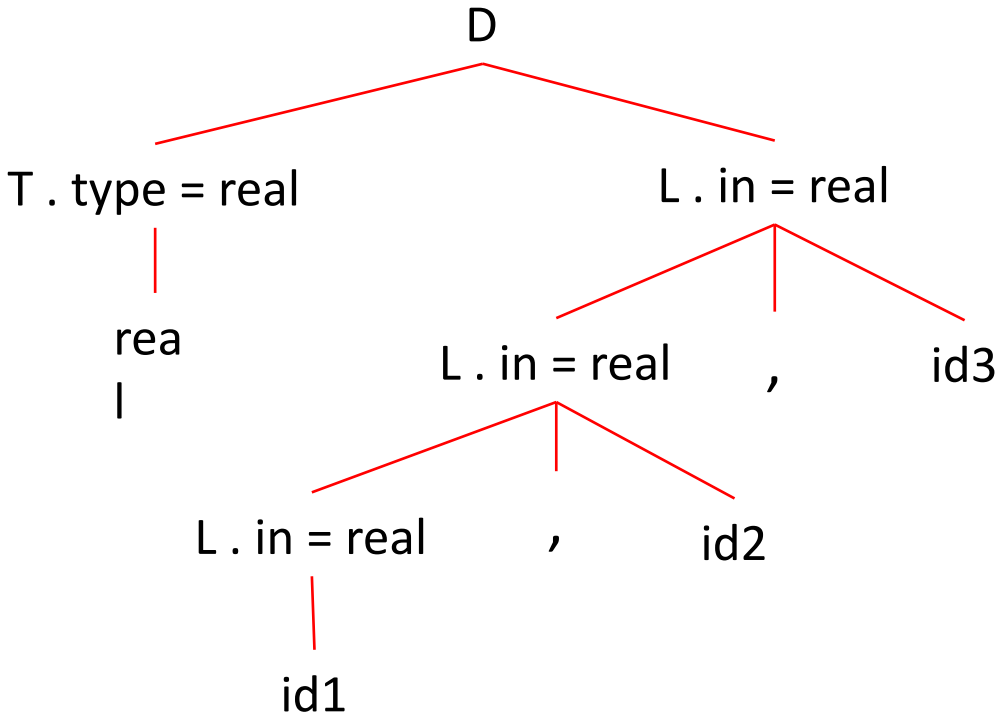
Make SDD to count number of 0's in a string

Make SDD to count number of 0's and 1's in a string

SDD (declaration of int & real)

Production	Semantic Rule
D → T L	L.in = T.type
T → **int**	T.type = integer
T → **real**	T.type = real
L → L , **id**	L.in = L.in addtype(id.entry , L.in)
L → **id**	addtype(id.entry , L.in)

Annotated parse tree for
real id1 , id2 , id3



Example of Inherited attribute

Dependency graph

- If an attribute “b” at a node in a parse tree depends on an attribute “c”,
  then the semantic rules for “b” at that node must be evaluated after the semantic rule that defines “c”

- The interdependencies among the inherited and synthesized attributes at that nodes in a parse tree can be shown using directed graph called “dependency graph”

- The graph has node for each attribute and
  an edge “c” to “b” if attribute “b” depends on attribute “c”

Dependency graph

- The dependency graph for a given parse tree is constructed as follow

for each node “n” in the tree **do**

  for each attribute “a” of the grammar symbol at “n” **do**

    construct a node in the dependency graph for “a”

for each node “n” in the pare tree **do**

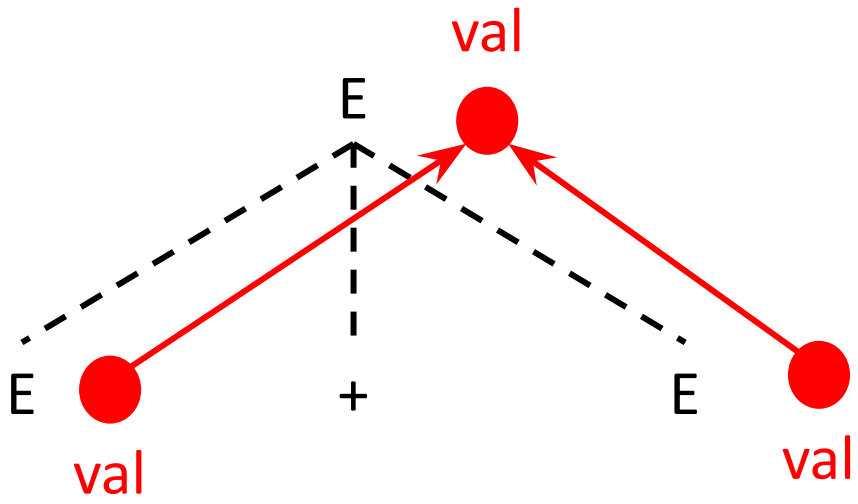
  for each semantic rule b = $f(c_1,c_2,\ldots c_k)$ associated with the production used at “n” **do**

    for i=1 **to** k **do**

      construct an edge from the node for “c_i” to the node for “b”

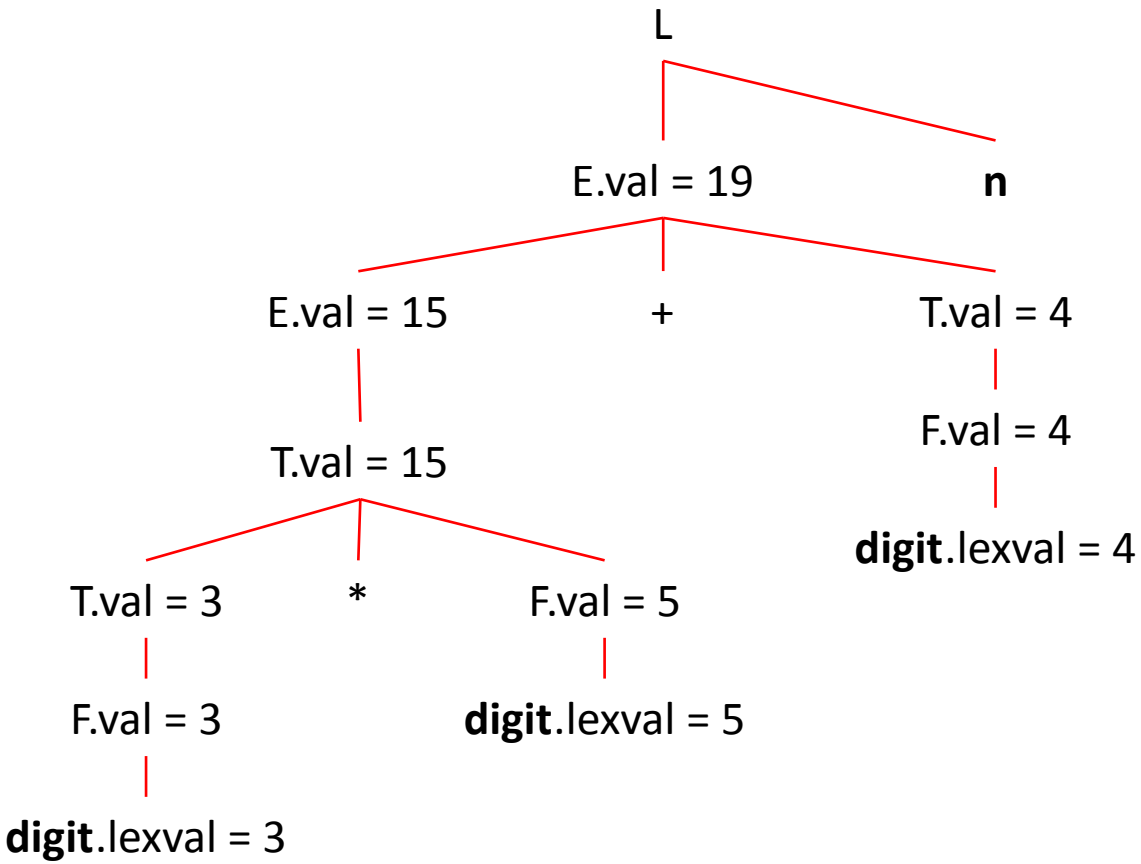
Dependency graph

Production	Semantic Rule
E → E + E	E.val = E.val + E.val

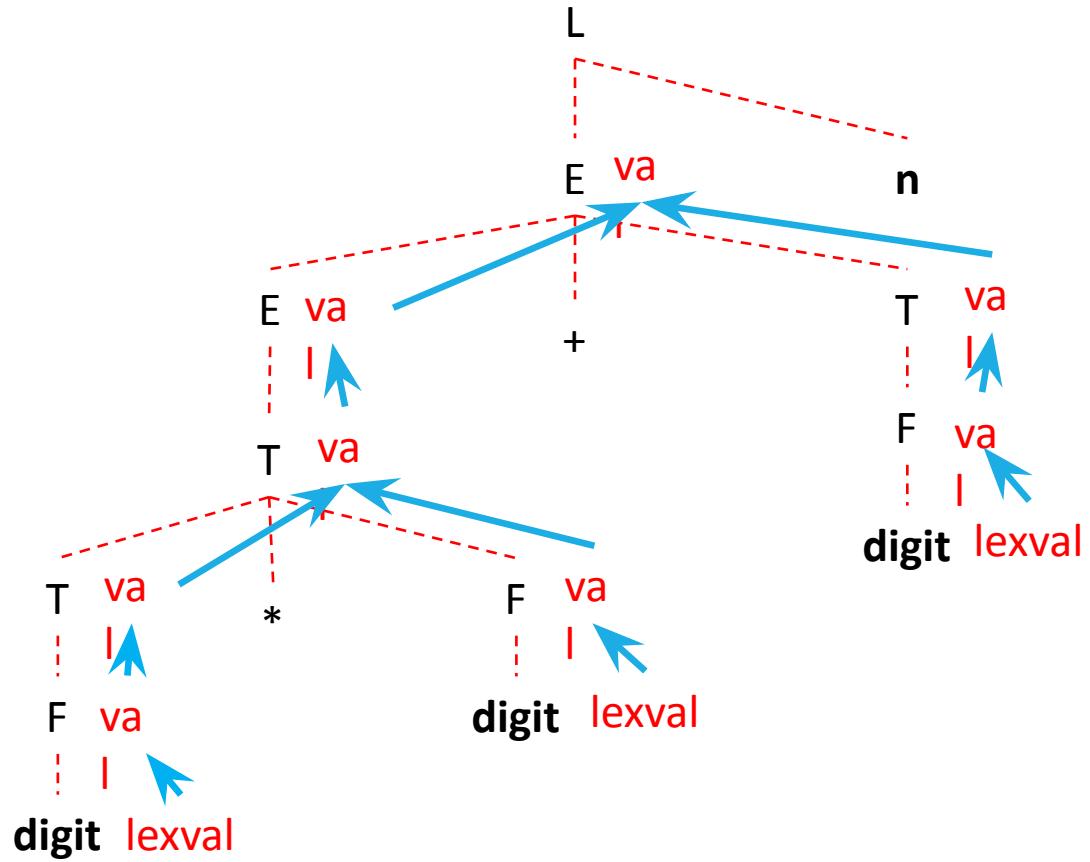


Dependency graph (Simple Desk Calculator)

Annotated tree

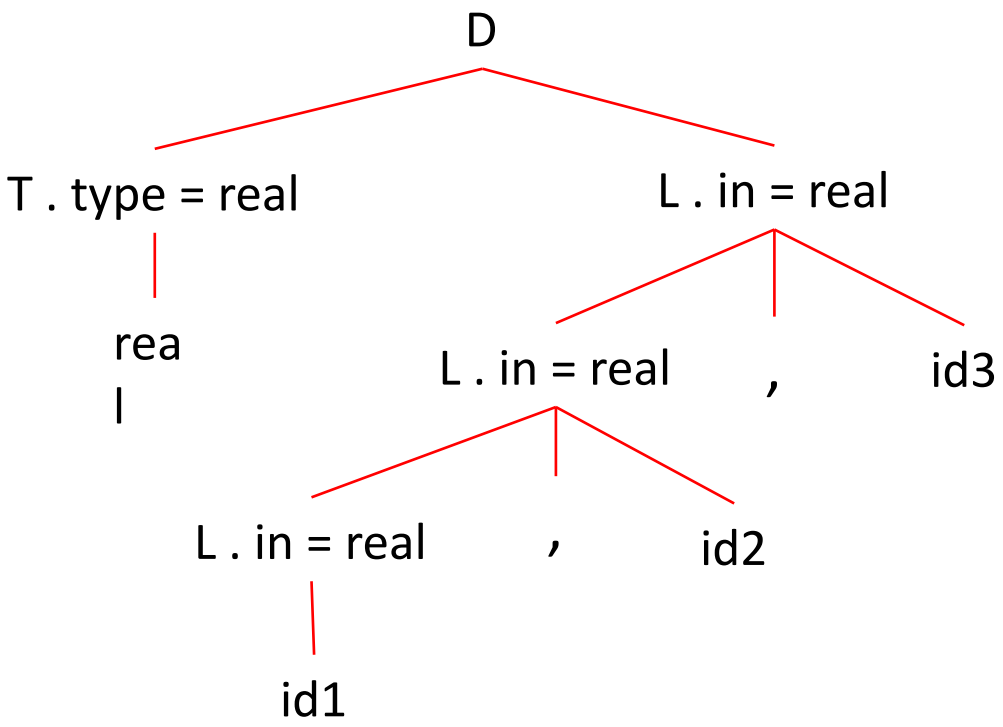


Dependency graph

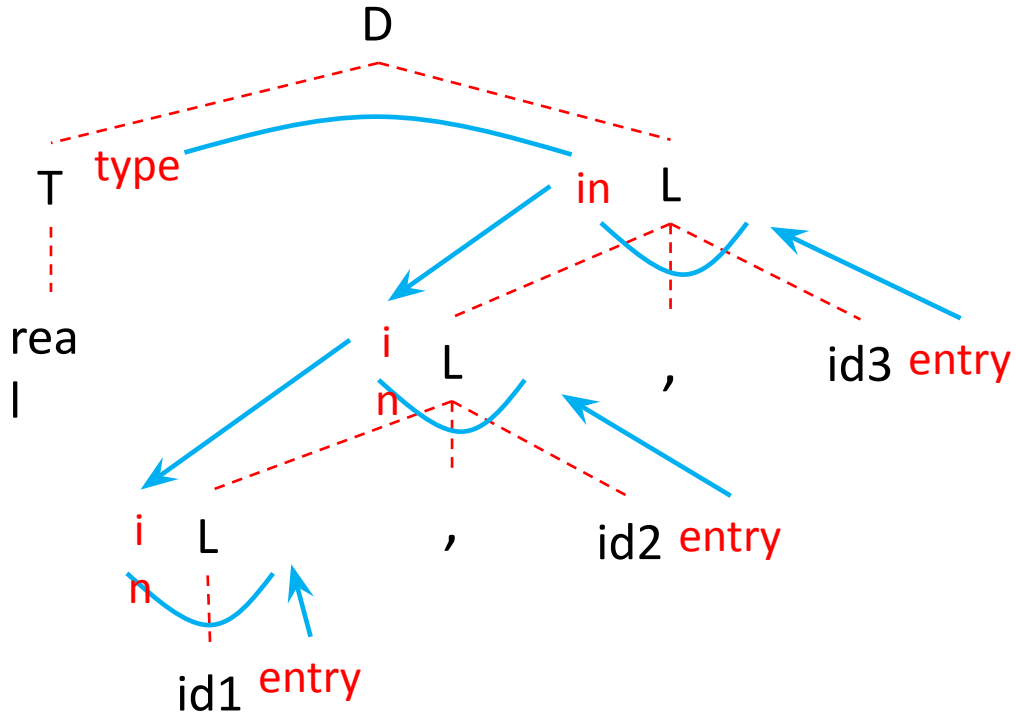


Dependency graph (declaration of int & real)

Annotated tree



Dependency graph



Topological sort

- A topological sort of a directed acyclic graph is any ordering m_1 , m_2 , … , m_k of the nodes of the graph such that edges go from nodes earlier in the ordering to late nodes
  i.e.
  if m_i ? m_j is the edge from m_i to m_j then
  m_i appear before m_j in the ordering

Evaluation order

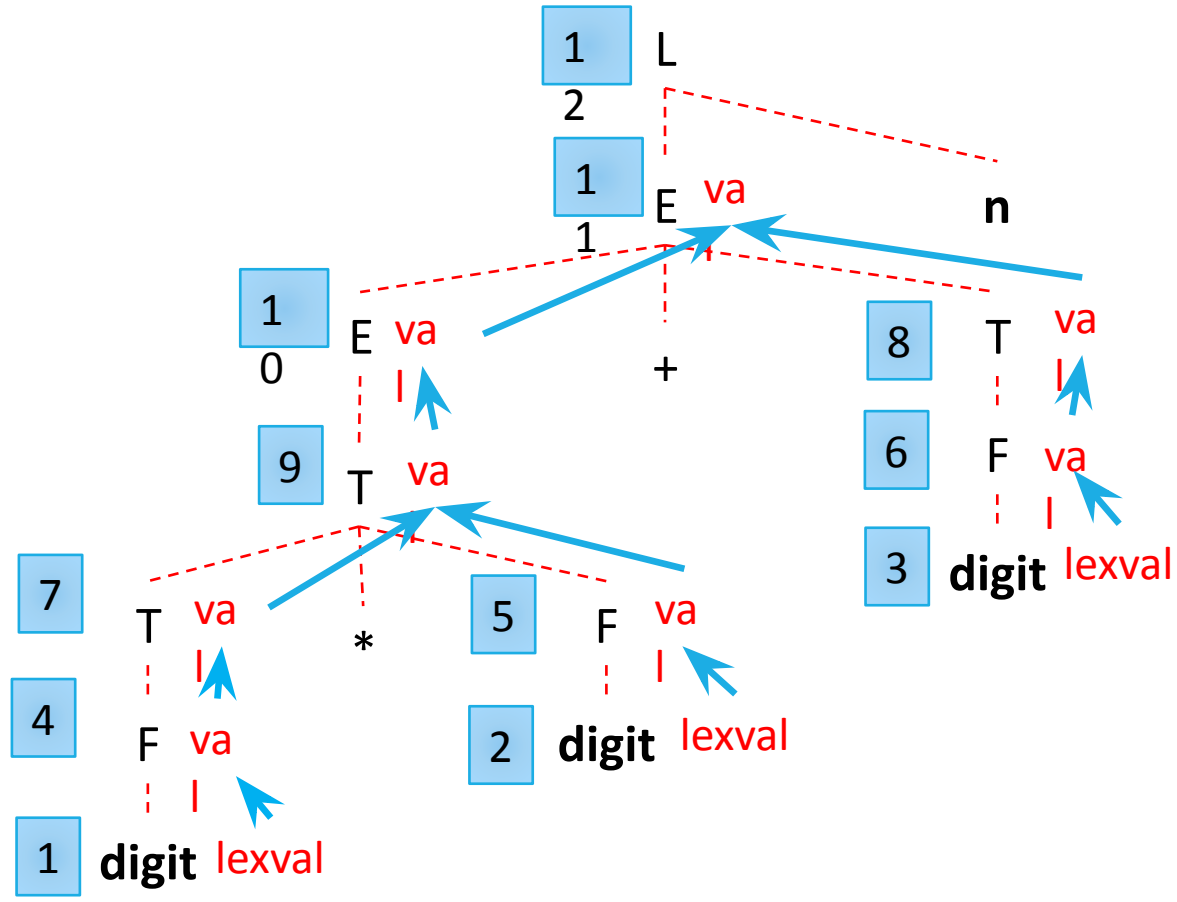
- From the topological sort of the dependency graph, we obtain an evaluation order for the semantic rule
- Evaluation of the semantic rules in this order yields the translation of the input string

Evaluation order (Simple Desk Calculator)

Evaluation order

a4 = a1
a5 = a2
a6 =a3
a7 = a4
a8 = a6
a9 = a7 * a5
a10 = a9
a11 = a10 + a8
Print (a11)

Dependency graph

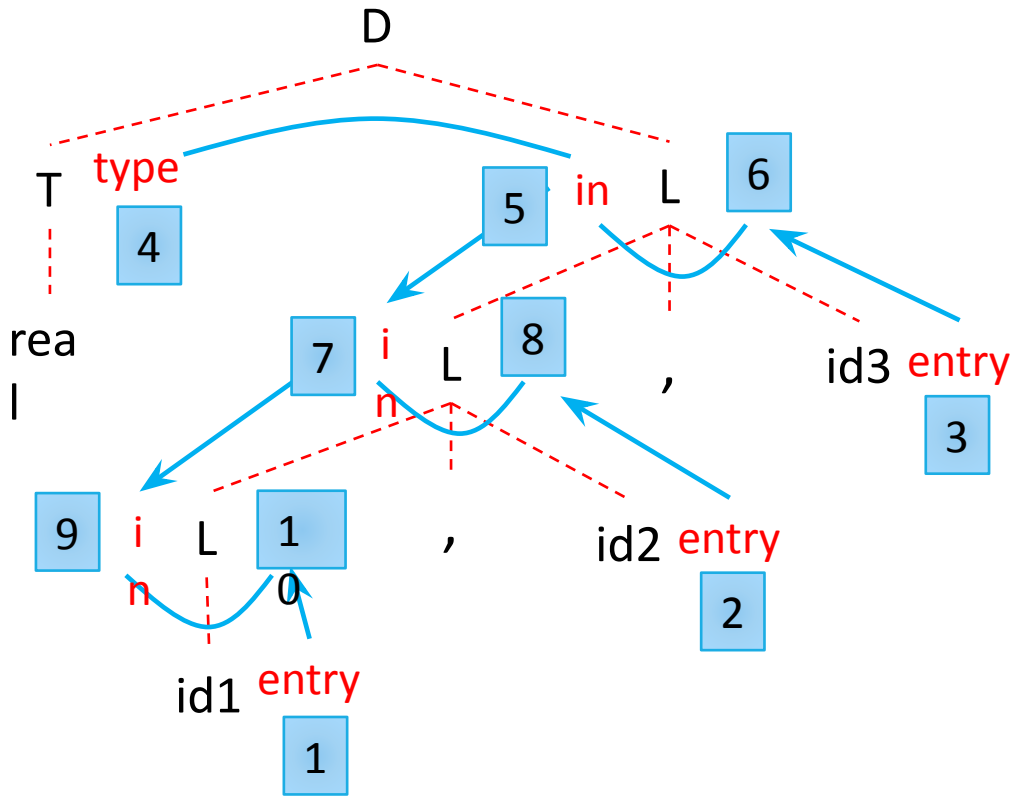


Evaluation order (declaration of int & real)

Evaluation order

a4 = real
a5 = a4
addtype (id3.entry , a5)
a7 = a5
addtype (id2.entry , a7)
a9 = a7
addtype (id1.entry , a9)

Dependency graph



Method for evaluating the semantic rules

Parse tree methods

Rule based methods

Oblivious methods

Method for evaluating the semantic rules

Parse tree methods

Rule based methods

Oblivious methods

- At compile time obtain an evaluation order from a topological sort of the dependency graph constructed from the parse tree for each input
- These methods will fail to find an evaluation order only if the dependency graph for the particular parse tree under consideration has a cycle

Method for evaluating the semantic rules

Parse tree methods

Rule based methods

Oblivious methods

- At compiler-construction time , the semantic rules associated with production are analyzed either y hand or by specialized tool
- For each production the order in which that production are evaluated is predetermined at compiler construction time

Method for evaluating the semantic rules

Parse tree methods

Rule based methods

Oblivious methods

- An evaluation order is chosen without considering the semantic rules
- E.g. if translation takes place during parsing then the order of evaluation is forced by the parsing method, independent of semantic rules

Method for evaluating the semantic rules

Parse tree methods

Rule based methods

Oblivious methods

- These methods dose not construct the dependency graph at compile time
- They can be more efficient in their use of compile time and space

Circular SDD

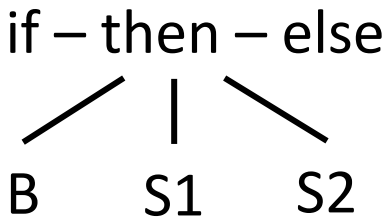
- A SDD is said to be circular if the dependency graph for some parse tree generated by its grammar has a cycle

Construction of Syntax Tree

- SDD can be used to specify the construction of syntax tree
- The use of syntax trees as an intermediate representation allows translation to be decoupled from parsing
- Translation routines (invoked during parsing) must live with two kinds of restrictions
  1. The grammar that is suitable for parsing may not reflect the natural hierarchical structure of constructs in language
  2. The parsing method constrains the order in which nodes in a parse tree are considered

Syntax tree

- An (abstract) syntax tree is a condensed (reduced) form of parse tree used for representing language constructs
- The production " S → **if** B **then** S1 **else** S2 "might appear in syntax tree as



- Operators and keywords do not appear as leaves,
  but associated with interior node that would be the parent of those leaves in the parse tree
- In syntax tree chain of single productions may be collapsed

Syntax tree

Grammar

L ? En

E ? E + T

E ? T

T ? T * F

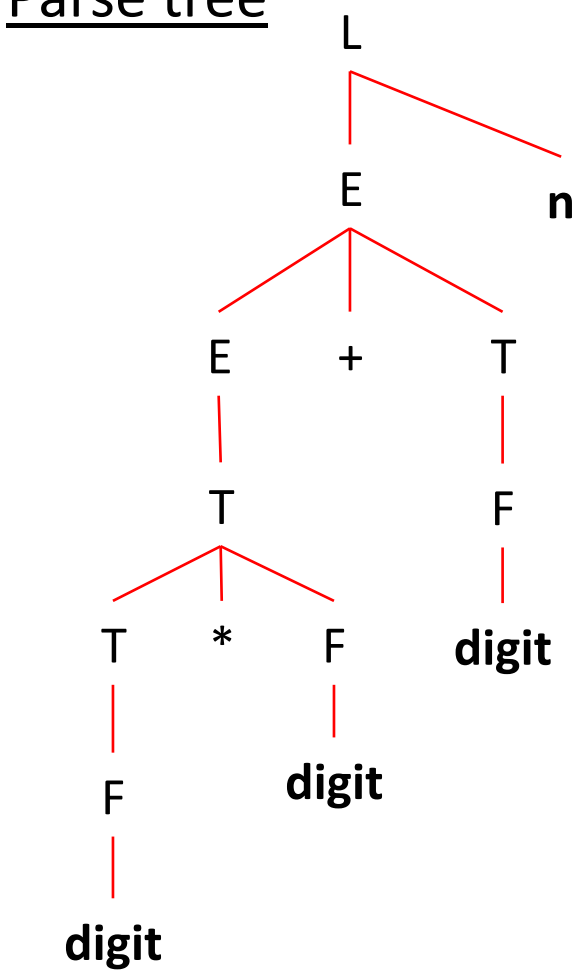
T ? F

F ? digit

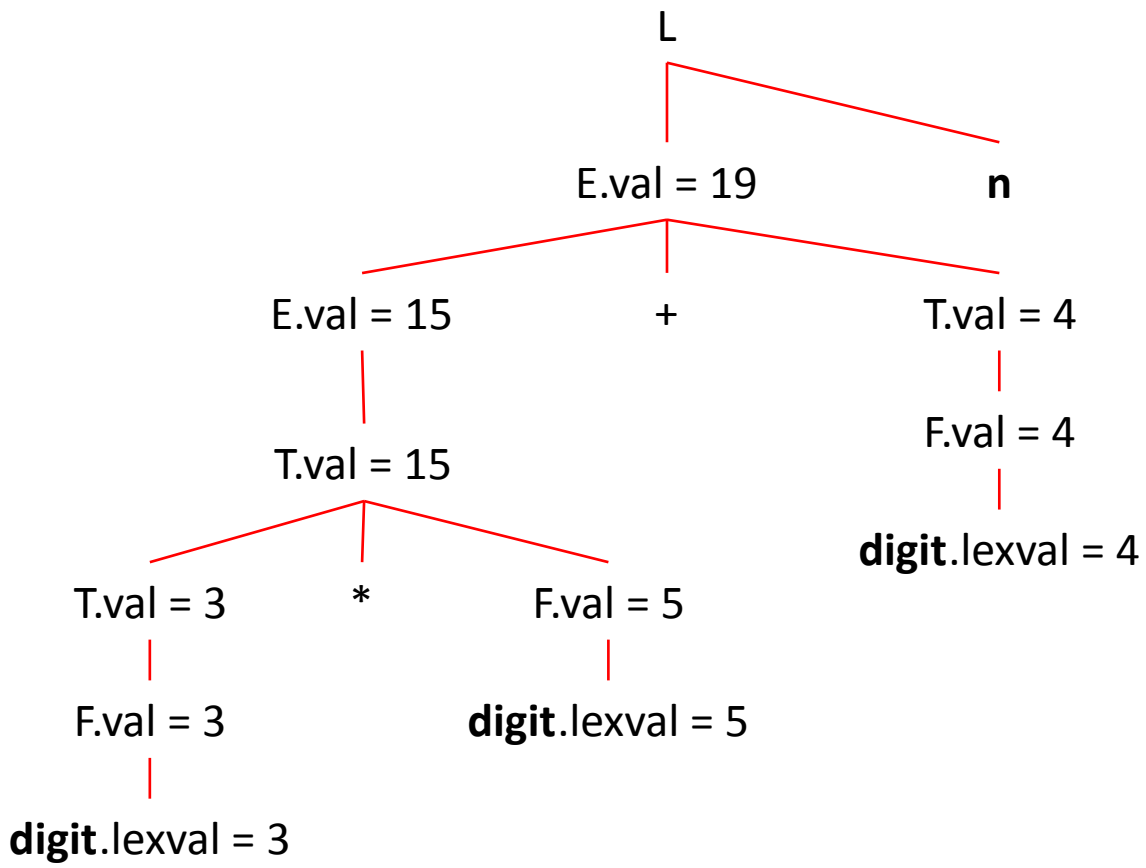
String

3 * 5 + 4

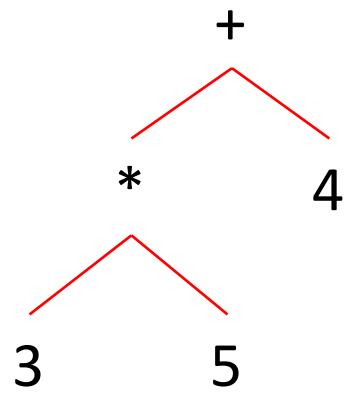
Parse tree



Annotated parse tree



Syntax tree



Constructing Syntax Tree for Expression

- Construct a subtree for each subexpression by creating a node for each operator and operand
- Children of an operator node are the roots of the nodes representing the subexpression constituting the operands of that operator
- Each node in the syntax tree can be implemented as a record with several fields

1. Node for OPERATOR
   - One filed is identify the operator
   - Remaining fields contain pointers to the nodes for the operand
2. Node for OPERAND
   - On field is identify operand (identifier/constant etc.)
   - Another field contain value or pointer to symbol table entry of that operand

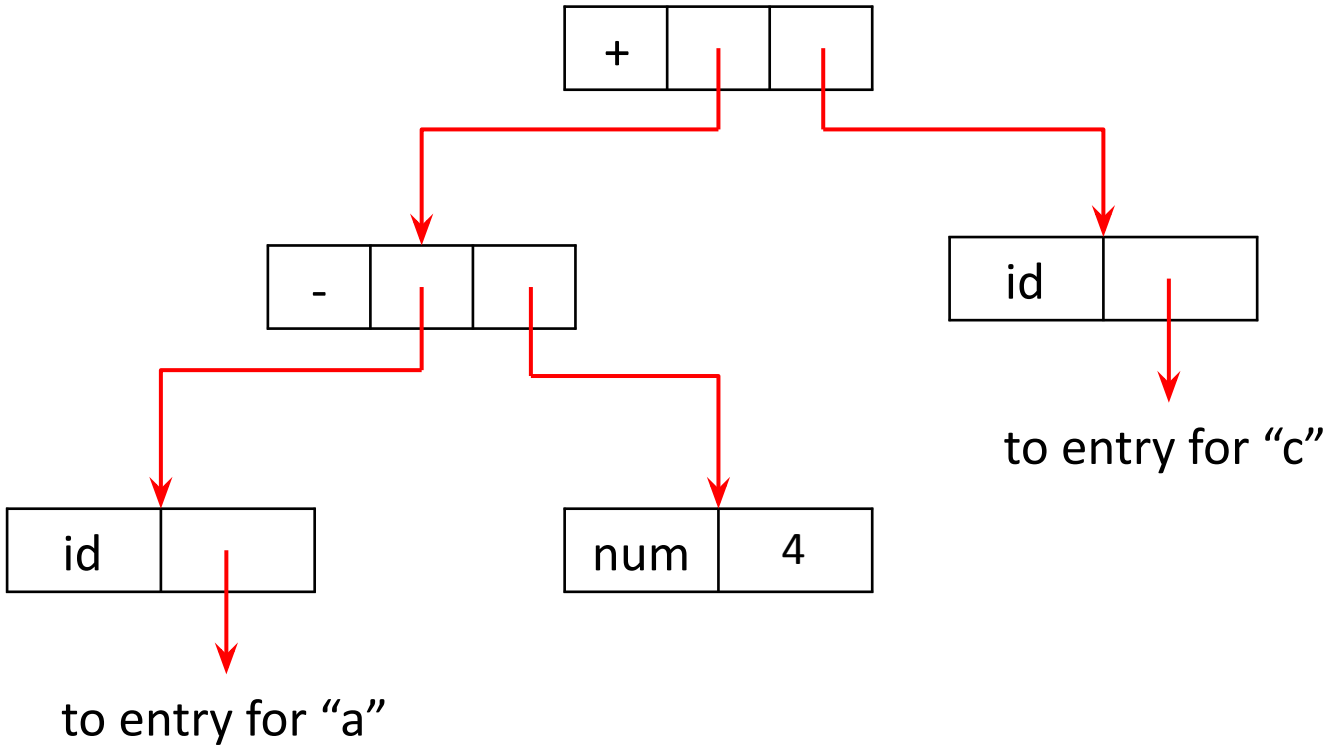
Constructing Syntax Tree for Expression

- Following functions are used to create syntax tree for expression with binary operator

1. *mknode (op , left , right)*
   - Creates an operator node with label op and two field contain pointers to left and right children
2. *mkleaf (**id** , entry)*
   - Creates an identifier nde with label is and a field containing pointer to symbol table entry for that identifier
3. *mkleaf (**num** , val)*
   - Creates a number node with label num and a filed containing value of that number

Constructing Syntax Tree for Expression

Construct Syntax Tree for : a – 4 + c

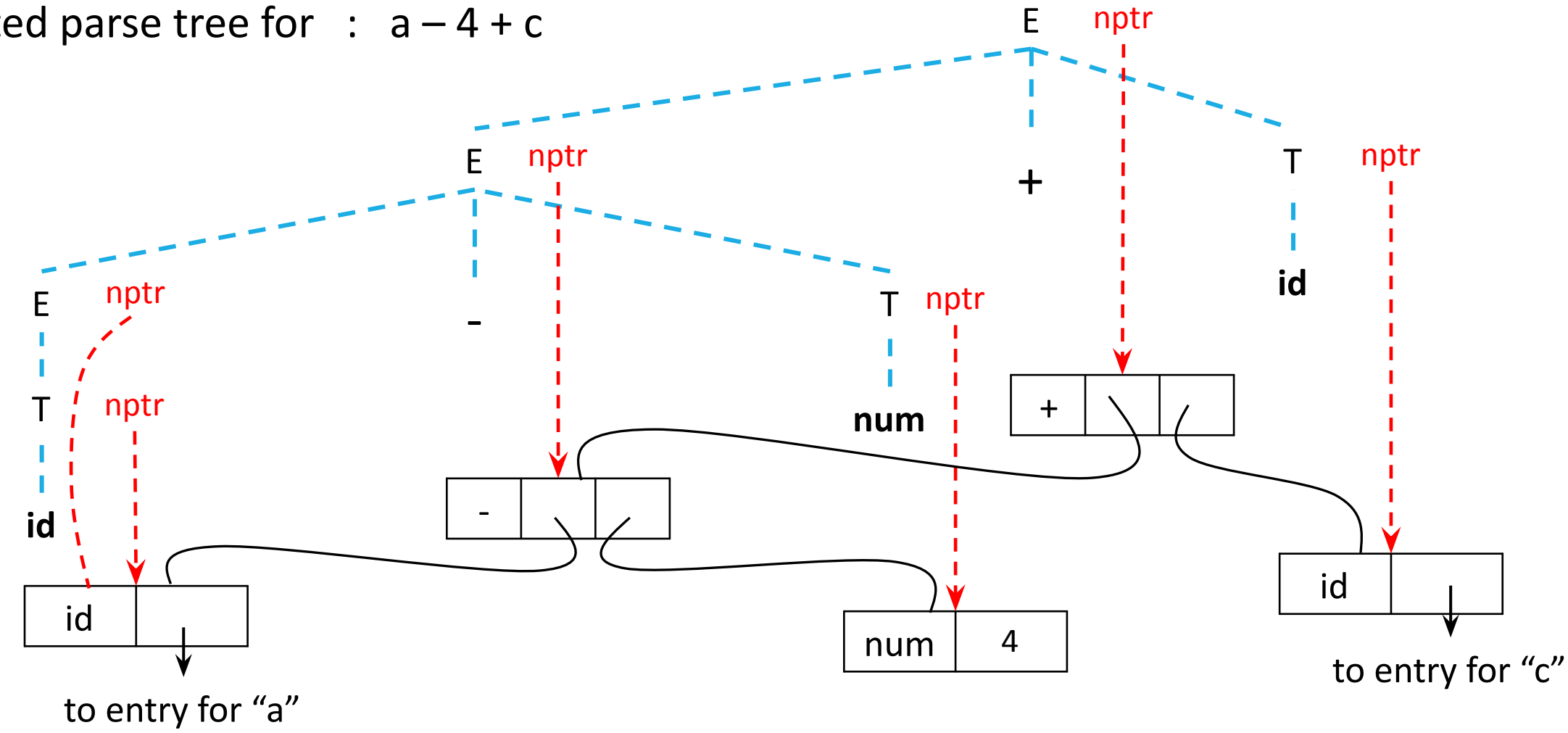


SDD (Constructing Syntax Tree for Expression)

Production	Semantic Rule
E → E + T	E.nptr = mknode (‘+’ , E.nptr , T.nptr)
E → E – T	E.nptr = mknode (‘-’ , E.nptr , T.nptr)
E → T	E.nptr = T.nptr
T → (E)	T.nptr = E.nptr
T → **id**	T.nptr = mkleaf (**id** , **id**.entry)
T → **num**	T.nptr = mkleaf (**num** , **num**.val)

Constructing Syntax Tree for Expression

Annotated parse tree for : a – 4 + c



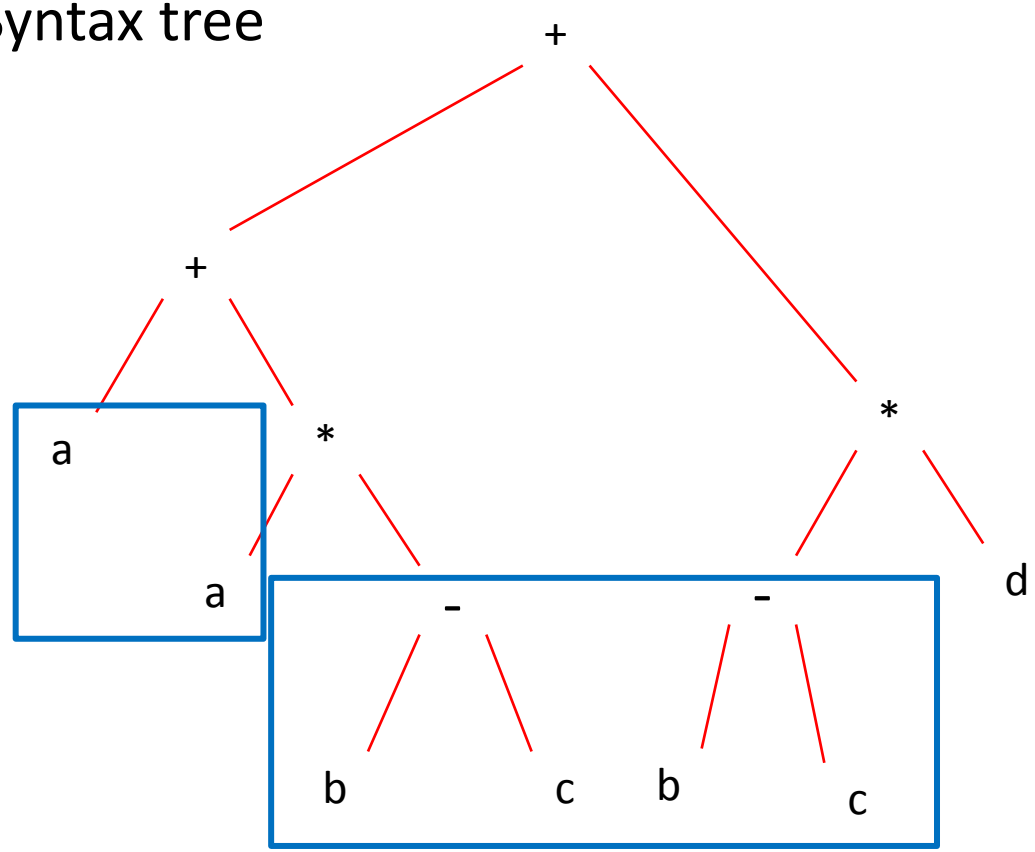
Directed Acyclic Graph (DAG) for Expression

- DAG for an expression identifies the common subexpression in the expression
- Like syntax tree, a DAG has a node for every subexpression of expression
  an interior node represents and operator and its children represent its operands
- Difference is that a node in a DAG representing a common subexpression has more than one parent in a syntax tree,
  the common subexpression would be represented as duplicated subtree

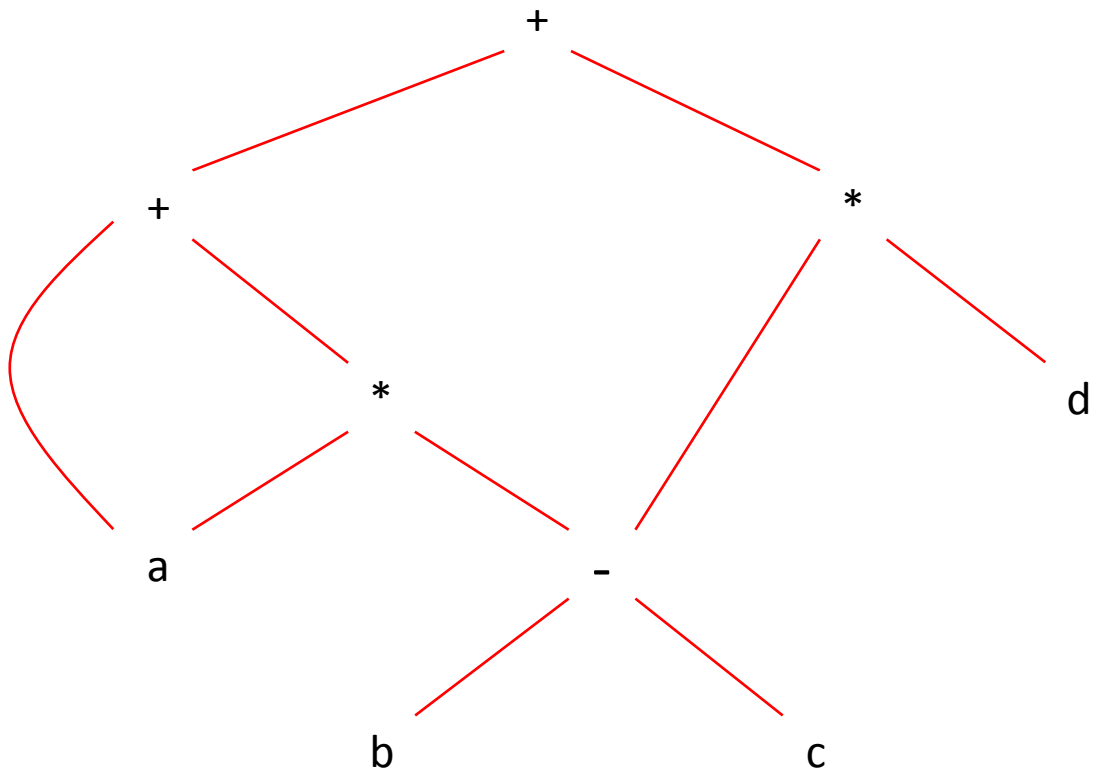
Directed Acyclic Graph (DAG) for Expression

a + a * (b – c) + (b – c) * d

Syntax tree



DAG



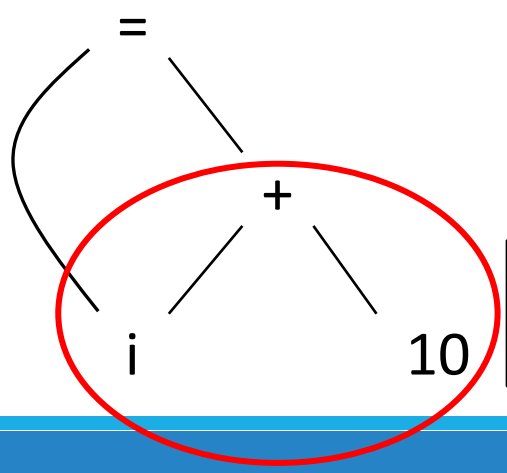
Value-number method

- In many applications, nodes are implemented as records stored in an array
- Each record has a label field that determines the nature of the node
- We can refer to a node by its index / position in the array
- The integer index of the node is often called a "value number"

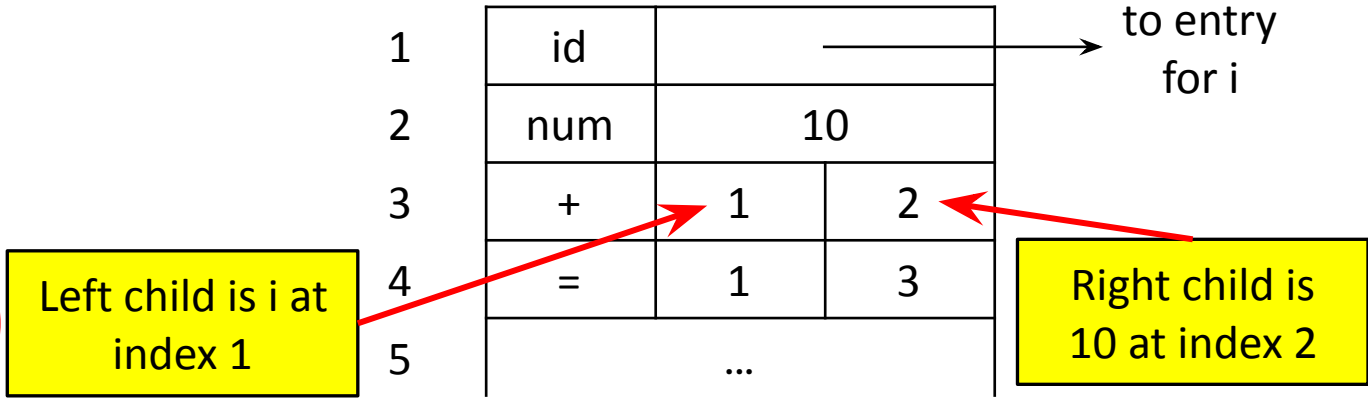
Assignment

i = i + 10

DAG



Representation



Value-number method

Algorithm

- Input
  - Label ***op***, node ***l*** and node ***r***
- Output
  - A node with signature <***op*** , ***l*** , ***r***>
- Method
  - Search the array for node ***m*** with label ***op***, left child ***l*** and right child ***r***
  - If there is such a node, return ***m***
    otherwise create a new node ***n*** with label ***op*** left child ***l*** right child ***r*** and return ***n***